Understanding Kalman Filters with Python

James Teow · Follow

14 min read · May 3, 2018

Listen Share More

Today, I finished a chapter from Udacity's Artificial Intelligence for Robotics. One of the topics covered was the Kalman Filter, an algorithm used to produce estimates that tend to be more accurate than those based on a single measurement alone.

While Thrun's course has been helpful, I found myself still unable to articulate how Kalman Filters work or why they are useful, so I decided to watch a series of lectures by Professor Biezen. I highly recommend going through them, as most of this writing is based on his lectures.

Why Kalman filters?

Imagine we are making a self-driving car and we are trying to localize its position in an environment. The sensors of the car can detect cars, pedestrians, and cyclists. Knowing the location of these objects can help the car make judgements, preventing collisions. But on top of knowing the location of the objects, the car needs to predict their future locations so that it can plan what to do ahead of time. For example, if it were to detect a child running towards the road, it should expect the child not to stop. The Kalman filter can help with this problem, as it is used to assist in tracking and estimation of the state of a system.

The car has sensors that determines the position of objects, as well as a model that predicts their future positions. In the real world, predictive models and sensors aren't perfect. There is always some uncertainty. For example, the weather can affect the

incoming sensory data, so the car can't completely trust the information. With Kalman filters, we can mitigate the uncertainty by combining the information we do have with a distribution that we feel more confident in.

Examining Errors

Usage of the Kalman filter acknowledges that some error and noise is implicit within both the prediction and measurement of the variables we're interested in tracking.

The filter does not assume all errors are Gaussian, but as cited from the Wikipedia description, the "filter yields the exact conditional probability estimate in the special case that all errors are Gaussian." Recall, that the Gaussian is characterized by two parameters: the mean (mu) and the width the of Gaussian, namely the variance (sigma squared).

$$c = \frac{1}{\sqrt{2\pi\sigma^2}}$$

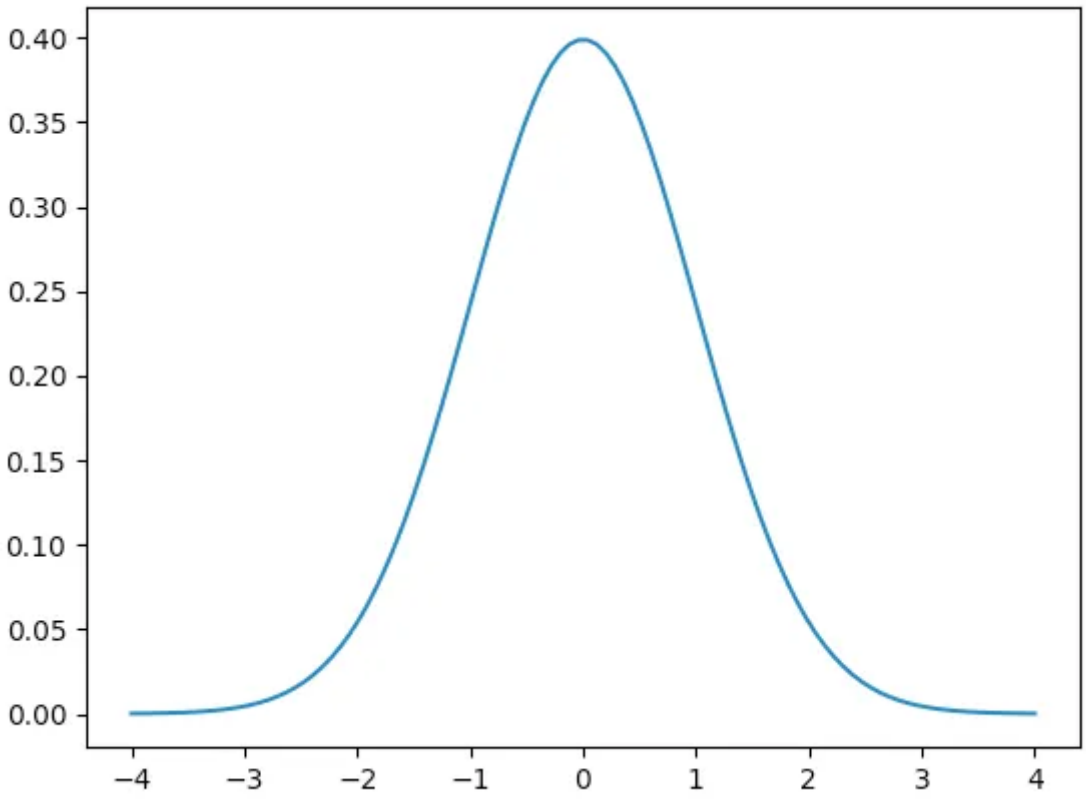
The constant that appears before the exponential.

$$g(x) = ce^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}$$

The quadratic difference between query point x relative to mean mu.

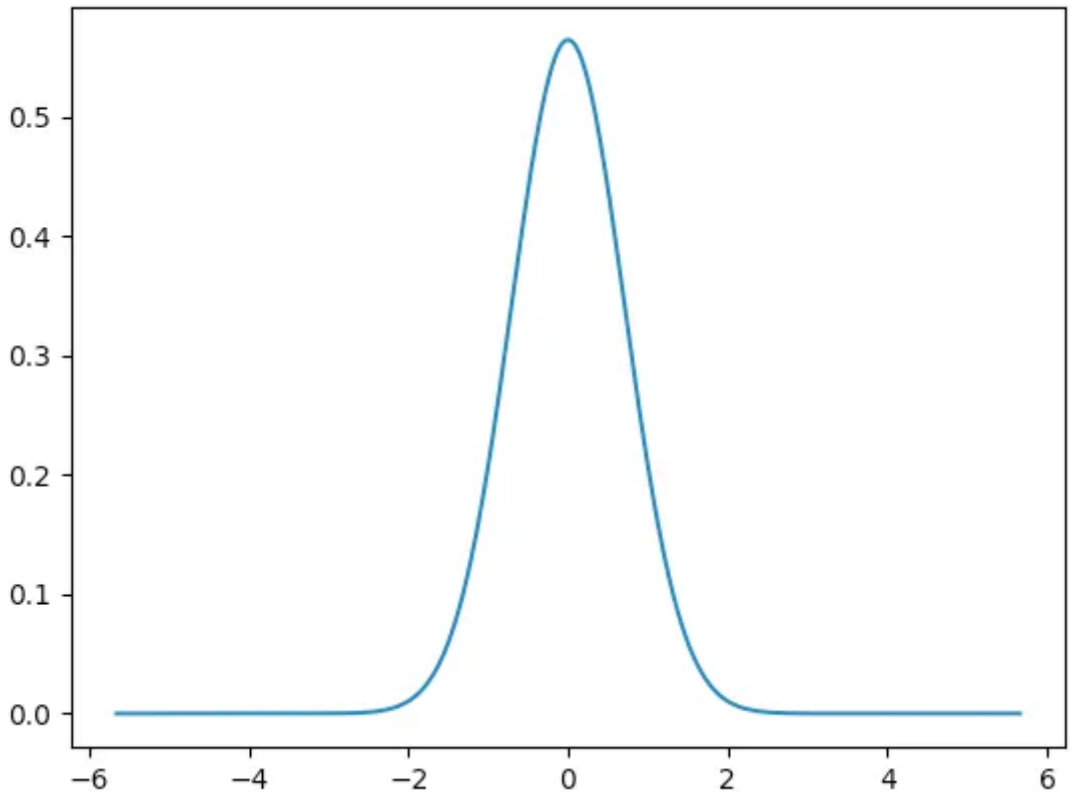
Instead of representing the distribution as a histogram, the task in Kalman filters is to maintain a mu and sigma squared as the best estimate of the location of the object we're trying to find. In the formula above, I summarized the normalizing constant as c because what's most important is noticing the relationship between the mean and the variance.

Notice, if x equals the mean of a distribution, the difference between the mean and the input is 0. We get e raise to power of 0, equaling 1.



Sample Gaussian distribution

We prefer a narrow Gaussian as it would have less variance, indicating more confidence in the data.



Sample Gaussian distribution with a narrow variance

The Kalman filter represents all distributions by Gaussians and iterates over two different things: **measurement updates** and **motion updates**. Measurement updates involve updating a prior with a product of a certain belief, while motion updates involve performing a convolution.

Measure Update

Measurement updates use Bayes Rule. Imagine we've localized another vehicle, and have a prior distribution with a very high variance (large uncertainty). If we get another measurement that tells us something about that vehicle with a smaller variance. If we create a new gaussian by combining the information, the mean will be somewhere in between the two distributions, with a higher peak and narrower variance than the prior.

It might be surprising that the subsequent Gaussian is peakier than the component Gaussian, but it makes some intuitive sense: by combining both, we've gained more

information than either Gaussian in isolation.

In the following equations, nu and r squared is respectively the mean and variance of new observed data. We can use it to update the prior mean and variance.

$$\mu' = \frac{r^2\mu + \sigma^2\nu}{r^2 + \sigma^2}$$

Update the prior mean with the weighted sum of the old means, where the weights are the variances of the other mean. The mean is normalized by the sum of the weighting factors.

$$\sigma^{2'} = \frac{1}{\frac{1}{r^2} + \frac{1}{\sigma^2}}$$

Update of the variance uses the previous variances.

It's interesting to note that if we had two Gaussian distributions with the same variance but different means, the resulting Gaussian distribution would be somewhere in the middle, but with a variance that's half of the original variance. We can see this calculating the resulting variance, which would be sigma squared over 2.

```python
def updated_mean(mean1, var1, mean2, var2):
    new_mean = (mean1 * var2 + mean2 * var1) / (var1 + var2)
    return new_mean

def updated_var(var1, var2):
    new_var = 1 / ((1 / var1) + (1 / var2))
    return new_var
```

update_mean_and_variance.py hosted with ❤ by GitHub view raw

Measurement Update step of mean and variance for a one dimension Kalman filter.

Prediction / Motion Update

For the prior, the car is believed to start in some position. But when it moves, it moves position. The motion itself has its own Gaussian distribution and uncertainty. We arrive at a prediction that adds the motion command to the mean, and has increased

uncertainty over the initial uncertainty. **We accumulate more uncertainty as we change position.**

$$\mu' = \mu + u$$

The mean is updated by taking the previous mean and adding the motion of the movement, indicated by variable u.

$$\sigma^{2'} = \sigma^2 + r^2$$

The variance is updated by taking the old sigma and adding onto it the variance of the motion Gaussian

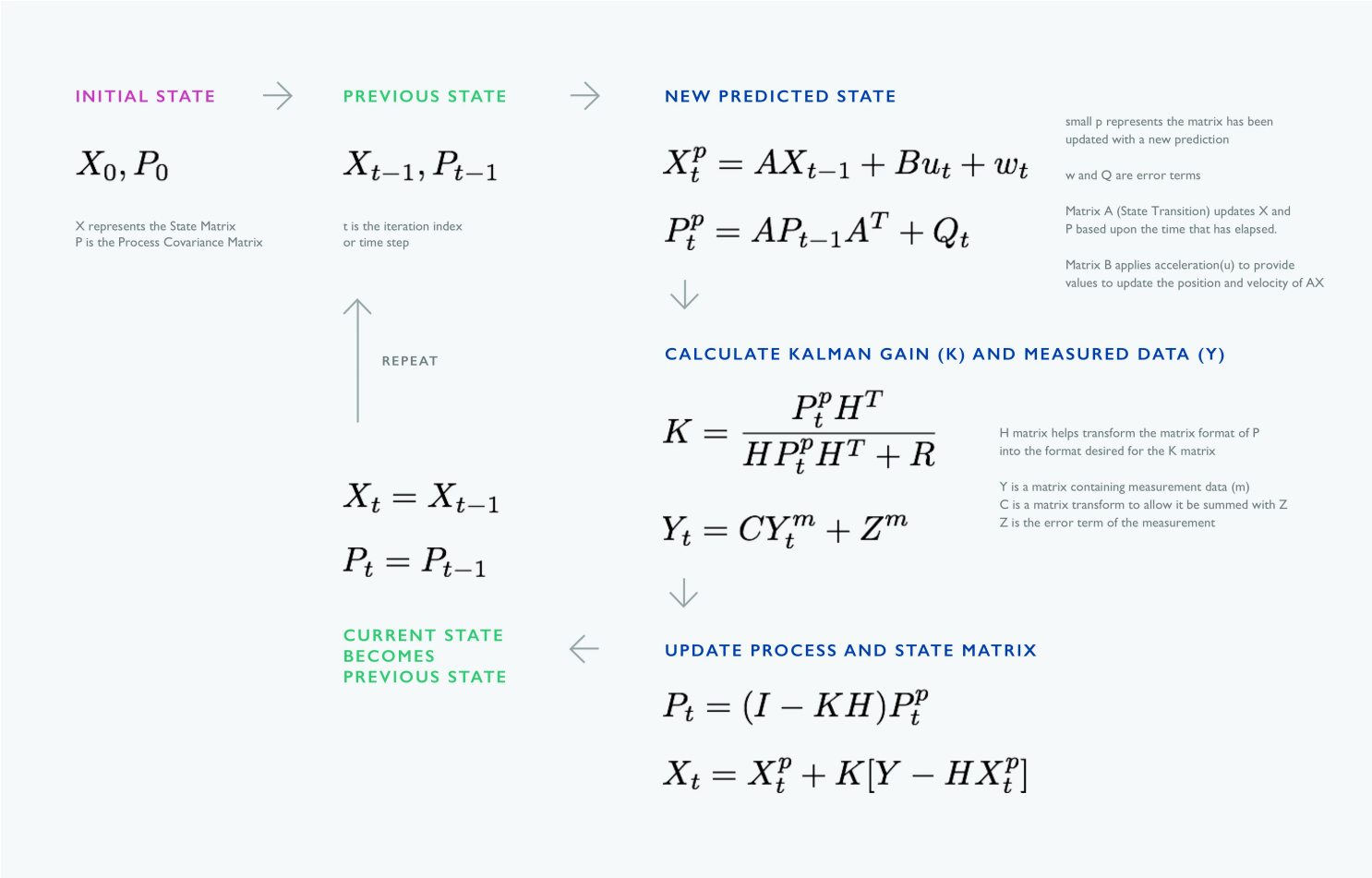
Notice that the variance update will always result in an increased variance.

```python
def predict(mean1, var1, mean2, var2):
    new_mean = mean1 + mean2
    new_var = var1 + var2
    return [new_mean, new_var]
```

kalman_filter_predict.py hosted with ❤️ by GitHub view raw

Prediction Update of a 1D Kalman Filter

Designing a Kalman Filter



Flowchart of a Kalman Filter Matrix process, inspired by Prof. Biezen's flowchart

The Kalman filter is a uni-modal, recursive estimator. Only the estimated state from the previous time step and current measurement is required to make a prediction for the current state.

The flowchart above shows the recursive process. From the beginning, the filter keeps track of State variable that contains the current value of the measurements being tracked (e.g. position, velocity), while the Process variable contains the predictive error of those measurements.

Each time step, the state transition matrix moves the state and process matrix based on the current position and velocity, estimating a new position/velocity as well as new covariance.

From there, the Kalman Gain is calculated, along with the observed data. The update process involves using the Kalman in conjuction with the previous estimate and new observed data to update the state variable towards a belief that's somewhere between

the prediction and measurement. The process covariance is also updated based on the Kalman gain. These updates are then used for the next round of predictions.

Kalman Gain

The Kalman gain is used to determine how much of the new measurements to use to update the new estimate.

To calculate the gain, we need two things. One, we need the error in the estimate (or the original error). Everytime we calculate the error in the estimate, we use that information to update the Kalman gain. Two, we need error in the data/measurement, because as we continually get data inputs into the estimate we need to determine how that affects the gain.

What the gain does is put a relative importance between the error in the estimate vs the error in the measurement. If the error in the estimate is smaller, we put more emphasis on it. Vice versa for the error in data. What feeds the overall calculation depends on how much we can trust the prediction and data (which we base on the error).

$$K = \frac{E_{est}}{E_{est} + E_{mea}}$$

Kalman Gain is calculated by comparing the error in the estimate relative to the error of the estimate and measurement combined

The number for the Kalman gain will be somewhere 0 and 1. It is then used to update the value of the current estimate. In the equation below, x represents the estimate, K is the Kalman gain, which is multiplied (acting like a weight) by the difference between the measurement (p) and the previous estimate.

$$x_t = x_{t-1} + K[p - x_{t-1}]$$

The Kalman gain is used to adjust the update to the state.

If the Kalman Gain is close to 1, it means the measurements are accurate but the estimates are unstable. We can infer this by looking at the Kalman Gain ratio, where if

the error of the measurement is small (and practically contributing nothing to E_est over E_st), then K will be equal to one. When the error of the measurement is small, future predictions will be strongly updated based on new input data.

However, if the Kalman Gain is small, then it the error in the estimate is large relative to the error in the estimate. Estimates are more stable and the measurements are inaccurate. As a result, any difference between new data and the prediction will have a smaller effect on the eventual update.

With each iteration, the model will make estimates closer to the true value resulting in a smaller Kalman Gain. New data could have a lot of uncertainty and we don't want it to throw off the predictive model.

$$Est_t = Est_{t-1} + K[M - Est_{t-1}]$$

Updating estimates weighted by Kalman gain

To recalculate the error in the estimate, we simply need to multiply the error of the measurement with the error of the previous estimate, and divide it by the sum of the errors.

The equation is sometimes written as follows:

$$E_t = [1 - K]E_{t-1}$$

Updating estimation errors

Where E is the error of the estimate and the Kalman Gain is multiplied by the previous estimate. The one minus K factor being multiplied with the previous error is the inverse of the size of the Kalman Gain. If the Kalman Gain is large, the error in the measurement is small, so new data can quickly update the model to the true value, which will subsequently reduce the error in the estimate. However, if the Kalman Gain is very small, error in the measurement must be very large, then updates should be slowly.

The Kalman Gain ultimately drives the speed in which the estimated value is zeroed in on the true value.

State Matrix

The state matrix records the object being tracked. It could be another car on the road or a plane in the air.

$$X_t^p = AX_{t-1} + Bu_t + w_t$$

State Matrix Prediction

The condition of the state is updated based upon the previous condition of the state plus the control variable matrix u, with w representing noise that may have accumulated through the process.

$$X = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \qquad X = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \end{bmatrix}$$

The state matrix can represent multi-dimensions multiple ways, but the ordering will affect how the A matrix is constructed.

The state matrix can contain information such as the position (x and y) and the velocity (x dot and y dot) of an object.

$$x = x_0 + \dot{x}t + \frac{1}{2}\ddot{x}t^2 \qquad d = d_0 + vt + \frac{1}{2}at^2$$

Updating the position involves determining the displacement of the object given the acceleration and velocity.

$$AX = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} x + \Delta T \dot{x} \\ \dot{x} \end{bmatrix}$$

First part of updating matrix X.

Matrix A times x represents the current state and velocity based on the next time step (delta t). A time step is taken, and the velocity is added onto the previous position to update the position of the object. The velocity remains the same. The velocity may have changed after the time step due to acceleration (control variable matrix). If there was acceleration, than this calculation isn't complete since the acceleration would've affected the velocity. This update however is applied in Matrix B times u.

$$Bu = \begin{bmatrix} \frac{1}{2}\Delta T^2 \\ \Delta T \end{bmatrix} \begin{bmatrix} a \end{bmatrix} = \begin{bmatrix} a\frac{1}{2}\Delta T^2 \\ a\Delta T \end{bmatrix}$$

Second part of updating matrix X.

The B matrix mimics part of the kinematics equation where the velocity and acceleration are multiplied by time. When matrix B is multiplied by the control variable (in this case, acceleration) and added to AX, it results in a change to the position and velocity due to acceleration.

Observation Data

Part of the Kalman filter process is imparting observation data with the state matrix containing the most recent prediction.

$$Y_t = CY_t^m + Z_t$$

Observation function.

The observation is equal to matrix C times the variables observed plus measurement noise. The shape and entries of matrix C is dependent on the number of variables we

want to observe. If we want to examine all the variables in Y, then C would largely just be an identity matrix. If however some variables of Y aren't being observed, C could be shaped in a way so as to discard some of the variables of Y.

For example, if the incoming data contains four entries (x, y, x velocity, y velocity) but we only want two (x and y), matrix C can be shaped to provide us only with the information that we want to retain (in this case, a 4 x 2 matrix with ones along the diagonal).

Z represents measurement noise.

State Covariance and Measurement Covariance Matrix

The state covariance matrix is an error of the estimate. The state matrices are estimations of the subsequent states. Since estimations have noise, errors, and uncertainties, Q, a process noise covariance matrix is added, which will adjust the state covariance matrix. It will be used to help the Kalman gain place emphasis on either the predicted value or the measured value.

$$P = \begin{bmatrix} \sigma_x^2 & \sigma_{x\dot{x}} \\ \sigma_{\dot{x}x} & \sigma_{\dot{x}}^2 \end{bmatrix}$$

Corresponding State Covariance Matrix

If the variance is set to 0, it means we have a fairly large certainty in the corresponding measurement. So for example, if the state covariance of the x position is 0, it means high confidence in the prediction of the x position.

The measurement covariance matrix (R) is the error of the measurement. Whenever a measurement is taken for the object that is being tracked, it doesn't mean that the measurement is exact, as there could be some error in the way the object is tracked.

$$P_k = AP_{k-1}A^T + Q$$

Where P is the state covariance matrix, Q is the process noise covariance matrix

$$K_k = \frac{P_k H^T}{H P_k H^T + R}$$

Kalman Gain as a matrix

Recall from earlier, if the measurement errors are small relative to the prediction errors, we want to put more trust in them (hence, the Kalman Gain will be closer to 1). Otherwise, if the measurement errors are larger than the prediction errors, the Kalman gain will put less emphasis on the difference between the prediction and the measurement.

If P is 0, then measurement updates are ignored. That would imply the predictive values are quite accurate and ignore the observations. It is highly unlikely that the model would have such exactitude, so one of the benefits of Q is that prevents P from being zero.

The A and H matrices are largely present to help format the matrices.

$$\sigma_x \sigma_y = \sum_{i=1}^{n} \frac{(\bar{x} - x_i)(\bar{y} - y_i)}{N}$$

Covariance, where x_bar is the mean of x

Covariance is a measurement of the joint variability of two random variables. If the variables tend to show similar behavior (e.g. the higher values of one variable correspond with the higher values of the other random variable), the covariance is positive. However, if the higher values of one variable correspond with the lesser values of the other variable, the covariance is negative. The sign is important because it indicates the tendency in the linear relationship between the variables.

The following is a simple 2x2 covariance matrix, where the variances of x and y are along the main diagonal, and the covariances occupy the upper and lower half of the diagonals. The variance of x could represent the variance in the velocity, while y could represent the variance in the position.

$$\begin{bmatrix} \sigma_x^2 & \sigma_x\sigma_y \\ \sigma_y\sigma_x & \sigma_y^2 \end{bmatrix}$$

Example 2 × 2 Covariance Matrix

$$\begin{bmatrix} \sigma_x^2 & \sigma_x\sigma_y & \sigma_x\sigma_z \\ \sigma_y\sigma_x & \sigma_y^2 & \sigma_y\sigma_z \\ \sigma_z\sigma_x & \sigma_z\sigma_y & \sigma_z^2 \end{bmatrix}$$

Example 3 × 3 Covariance Matrix

The variance, both plus and minus, should provide us the range of possibilities within the distribution. It also informs us how far from the mean we could anticipate a measurement, almost guaranteeing that no value will beyond the range of the mean minus/plus the variance.

To practice computing a covariance matrix, I produced the following code which would solve the sample problem as provided in this lecture where Professor Michel van Biezen gives an example of analyzing the variance and covariance of students scores. The rows of the input data represents sets of scores from students, with each column grouped by subject matter.

The dot product between the ones matrix and A matrix will result in a 5 x 3 matrix where each column contains the total of all marks accumulated for each subject, which

is then divided by the total number of tests taken for each subject, essentially providing the mean grade for each subject. The mean is then subtracted from the A matrix, producing the deviation.

The dot product of A transpose A produces the covariance matrix.

```python
import numpy as np

A = np.array([[90, 80, 40],
              [90, 60, 80],
              [60, 50, 70],
              [30, 40, 70],
              [30, 20, 90]])

ones = np.ones([5, 5])
deviation = A - (ones.dot(A) / len(A))
covariance = deviation.T.dot(deviation)
```

example_covariance_matrix_student_scores.py hosted with ❤️ by GitHub view raw

```
# Resulting Output
# Math, Physics English
[[ 3600.  2400. -1200.]
 [ 2400.  2000. -1400.]
 [-1200. -1400.  1400.]]
```

Additionally, he provided another example to work through how to create a covariance matrix for an state value.

```python
import numpy as np

# Initial Values
x = 50
x_dot = 5

# Standard Deviations
x_std = 0.5
x_dot_std = 0.2

# Variance
x_var = x_std ** 2
x_dot_var = x_dot_std ** 2

x_cov_x_dot = x_std * x_dot_std
x_dot_cov_x = x_dot_std * x_std

X = np.array([[x], [x_dot]])
P = np.array([[x_var, x_cov_x_dot],
              [x_dot_cov_x, x_dot_var]])
```

state_covariance_matrix.py hosted with ❤️ by GitHub view raw

```
# Resulting State Covariance Matrix:
[[0.25 0.1 ]
 [0.1  0.04]]
```

If the off diagonal terms were zero, it would indicate that the estimation error that contributes to one variable is independent of the other variable. In such a case, no adjustments are made to the estimates of one variable due to the estimation error of the other variable. When this is multiplied with other matrices such as updating the state and Kalman gain, the covariances won't influence the process.

Putting it all together by tracking an aircraft with a Kalman Filter

For the final problem, Professor Biezen provided the scenario of trying to determine the position and velocity of an aircraft. For the purposes of simplicity, the problem involves only the x position and x velocity. He also zeroed out the off-diagonal values in covariance matrices, which I have also done in my code.

```
x_observations = np.array([4000, 4260, 4550, 4860, 5110])
v_observations = np.array([280, 282, 285, 286, 290])

z = np.c_[x_observations, v_observations]

# Initial Conditions
a = 2  # Acceleration
v = 280
t = 1  # Difference in time

# Process / Estimation Errors
error_est_x = 20
error_est_v = 5

# Observation Errors
error_obs_x = 25  # Uncertainty in the measurement
error_obs_v = 6

def prediction2d(x, v, t, a):
    A = np.array([[1, t],
                  [0, 1]])
    X = np.array([[x],
                  [v]])
    B = np.array([[0.5 * t ** 2],
                  [t]])
    X_prime = A.dot(X) + B.dot(a)
    return X_prime


def covariance2d(sigma1, sigma2):
    cov1_2 = sigma1 * sigma2
    cov2_1 = sigma2 * sigma1
    cov_matrix = np.array([[sigma1 ** 2, cov1_2],
                           [cov2_1, sigma2 ** 2]])
    return np.diag(np.diag(cov_matrix))


# Initial Estimation Covariance Matrix
P = covariance2d(error_est_x, error_est_v)
A = np.array([[1, t],
              [0, 1]])

```

```
# Initial State Matrix
X = np.array([[z[0][0]],
              [v]])
n = len(z[0])

for data in z[1:]:
    X = prediction2d(X[0][0], X[1][0], t, a)
    # To simplify the problem, professor
    # set off-diagonal terms to 0.
    P = np.diag(np.diag(A.dot(P).dot(A.T)))

    # Calculating the Kalman Gain
    H = np.identity(n)
    R = covariance2d(error_obs_x, error_obs_v)
    S = H.dot(P).dot(H.T) + R
    K = P.dot(H).dot(inv(S))

    # Reshape the new data into the measurement space.
    Y = H.dot(data).reshape(n, -1)

    # Update the State Matrix
    # Combination of the predicted state, measured values, covariance matrix and Kalman Ga
    X = X + K.dot(Y - H.dot(X))

    # Update Process Covariance Matrix
    P = (np.identity(len(K)) - K.dot(H)).dot(P)

print("Kalman Filter State Matrix:\n", X)
```

kalman_filter_tracking_plane.py hosted with ❤️ by GitHub view raw

problem, there's probably more matrix rotation than what's required, but I believe it's meant to make the formula invariant to different matrix sizes. There is no division in matrix operations, so to find the ratio I used the dot product with the inverse of what would otherwise be the denominator.

Notice that in matrix format, the Kalman gain is a matrix of the same dimension as the inputs, and along the diagonal are weights that adjust the observed position and velocity. The lower the weights, the lower the model trusts the observations compared to the predictions.

With the newly calculated Kalman gain, I used it to weigh the difference between the observation data with the prediction, which was then used to update the state matrix. I also used the Kalman gain to update the process covariance matrix.

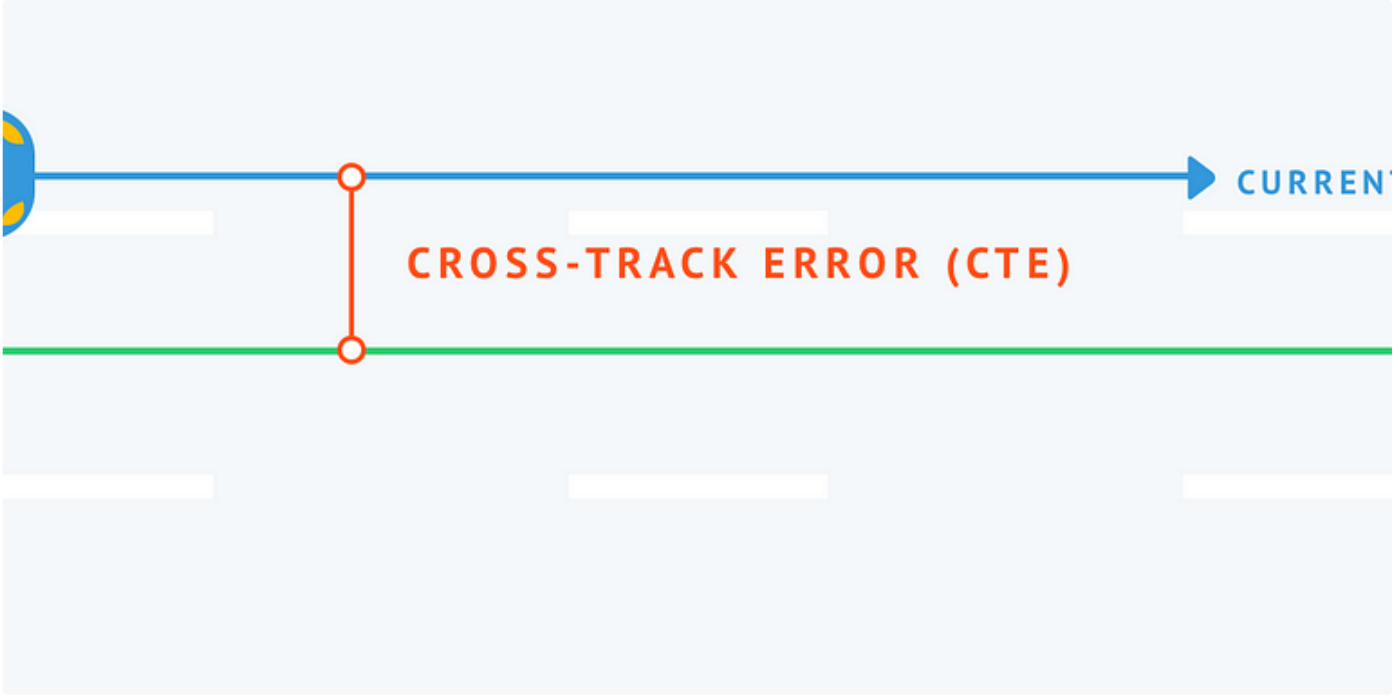
I did that as many times as observations that were provided, and got a result that is somewhere between the observation data and predictive model, which is exactly what one should expect from using the Kalman filter. My exact results were slightly different than Professor Biezen, but note that he did commit a number of errors in his calculations and did not do a full run through of all the observations, so I'm confident my calculations were more accurate.

Written by James Teow

267 Followers

¯_(ツ)_/¯

More from James Teow



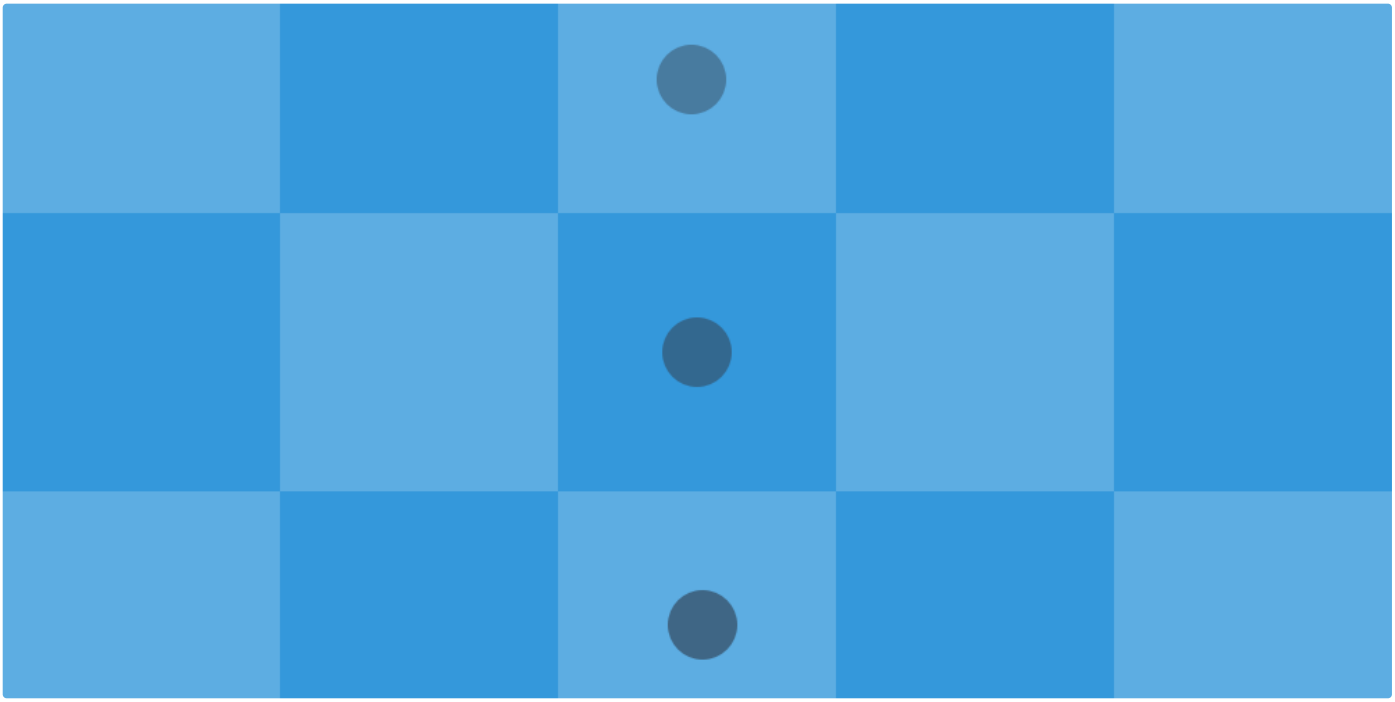
James Teow

Understanding Robot Motion: PID Control

This is part of a series of articles related to working through the AI for Robotics course on Udacity. This write-up is about Lesson 5.

7 min read · May 31, 2018

122 1



James Teow

Understanding Robot Motion: Path Smoothing

This is part of a series of articles related to working through the AI for Robotics course on Udacity. This write-up is about Lesson 5.

5 min read · May 26, 2018

140 1

James Teow

Gambler's Problem

Our agent is playing a game where they can place a bet on whether a coin flip will show heads. If it is heads, then they win the same...

8 min read · Dec 12, 2017

52 3

James Teow

Disassembling Jack's Car Rental Problem

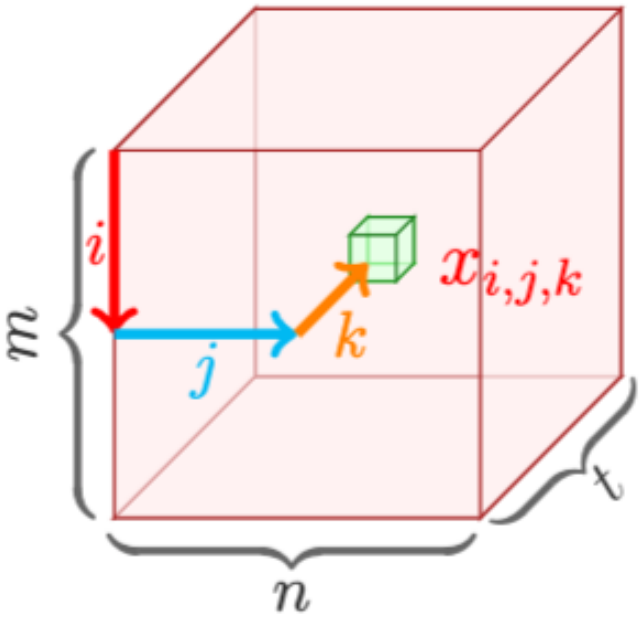
This is start of my exploration into learning about Reinforcement Learning. My hope is that by disassembling these exercises and...

16 min read · Dec 10, 2017

101 1

See all from James Teow

Recommended from Medium



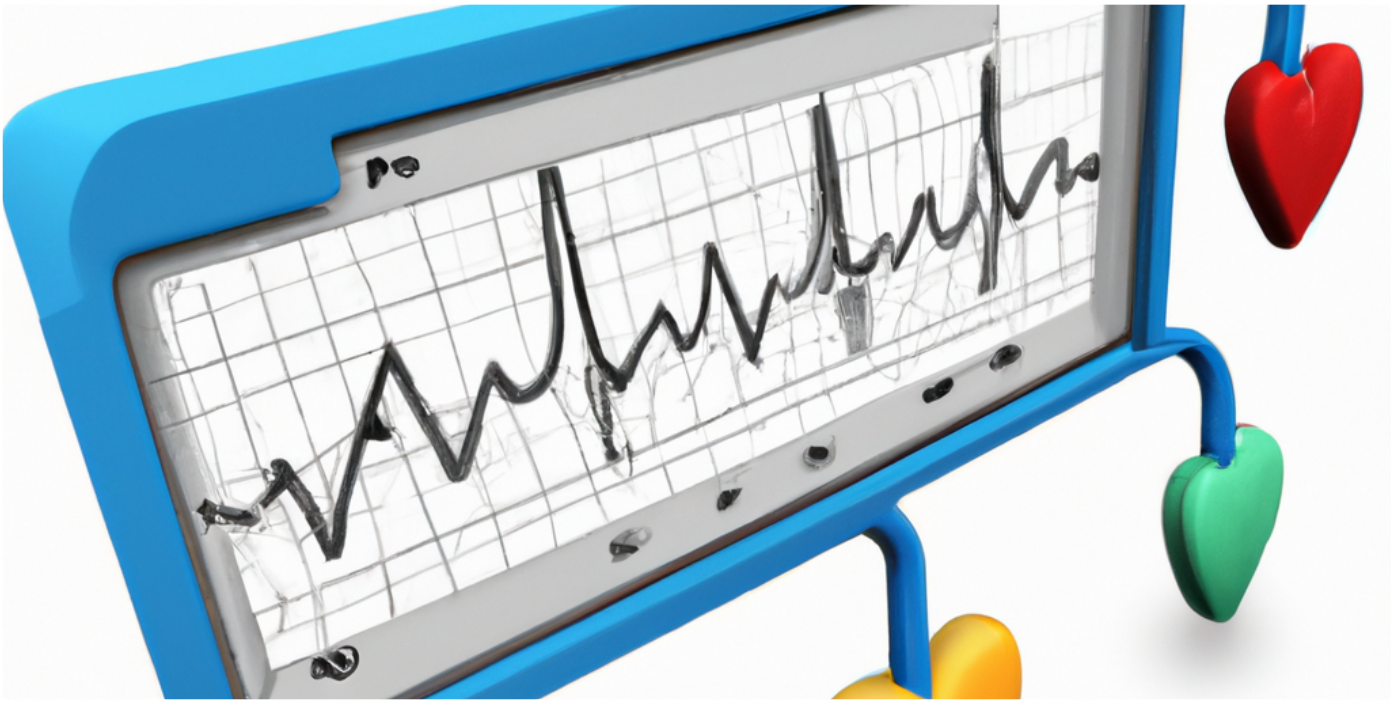
Xinyu Chen (*陈新宇*)

Intuitive Understanding of Tensors in Machine Learning

Tensor is an important concept in many scientific fields, such as mathematics, physics, signal processing, and computer vision, to name...

· 5 min read · Jan 20

12 1



Moklesur Rahman

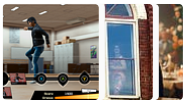
Denoising ECG Signal with Python Implementation

Denoising electrocardiogram (ECG) signals refers to the process of removing noise from ECG signals to improve the accuracy and...

· 5 min read · Jan 27

36

Lists

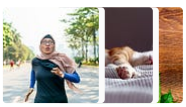


Staff Picks

338 stories · 94 saves

Stories to Help You Level-Up at Work

19 stories · 72 saves



Self-Improvement 101

20 stories · 127 saves

Productivity 101

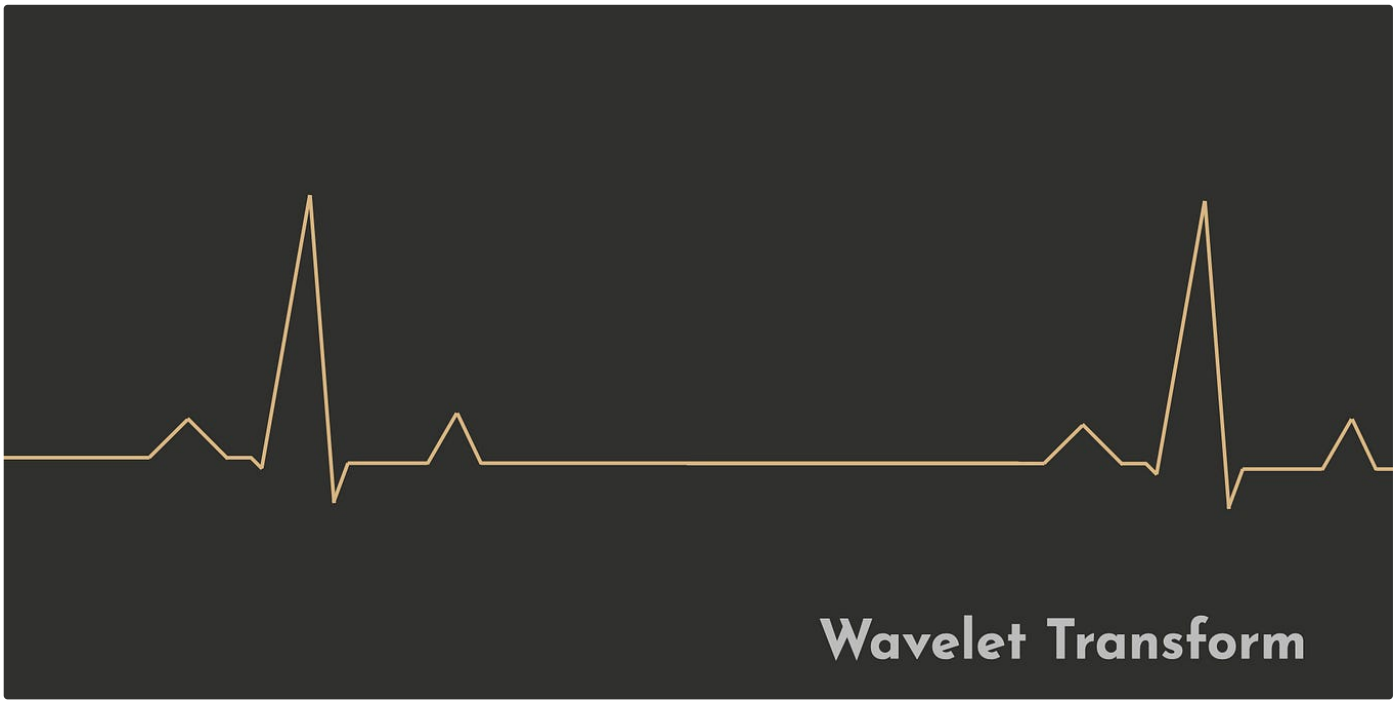
20 stories · 141 saves

Jerren Gan in Level Up Coding

How to Code and Apply a Kalman Filter on a 2-Dimensional System

Mastering an important tool for building autonomous systems

· 12 min read · Jan 24



Shawhin Talebi in Towards Data Science

The Wavelet Transform

An Introduction and Example

· 6 min read · Dec 20, 2020

456 4

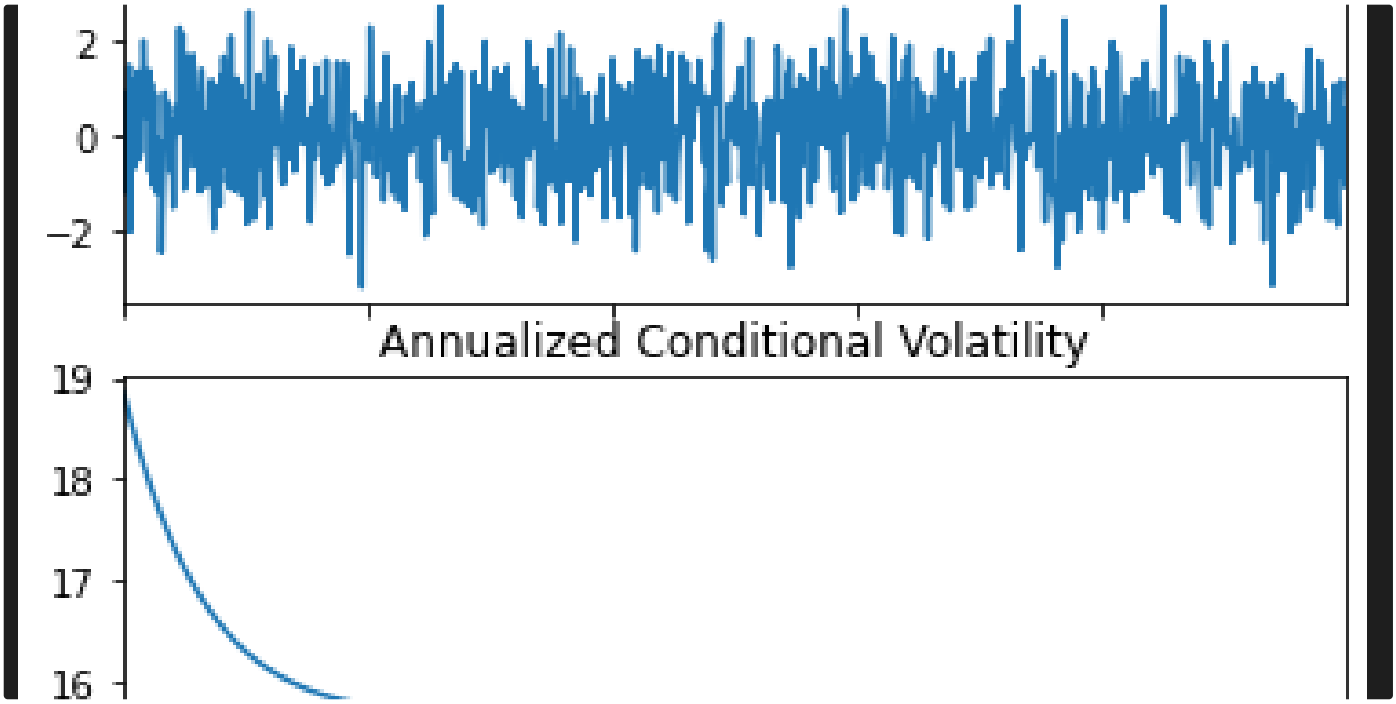
Louis Chan in Towards AI

Comprehensive Guide to Hampel Filter for Outlier Detection

Step by Step Walkthrough with S&P 500 Index

· 9 min read · Mar 12

61



Futuris Perpetuum

Building a GARCH Volatility Model in Python: A Step-by-Step Tutorial with Statistical Analysis

The Generalized Autoregressive Conditional Heteroscedasticity (GARCH) model is a statistical model that is widely used to analyze and...

· 3 min read · Feb 23

59

See more recommendations